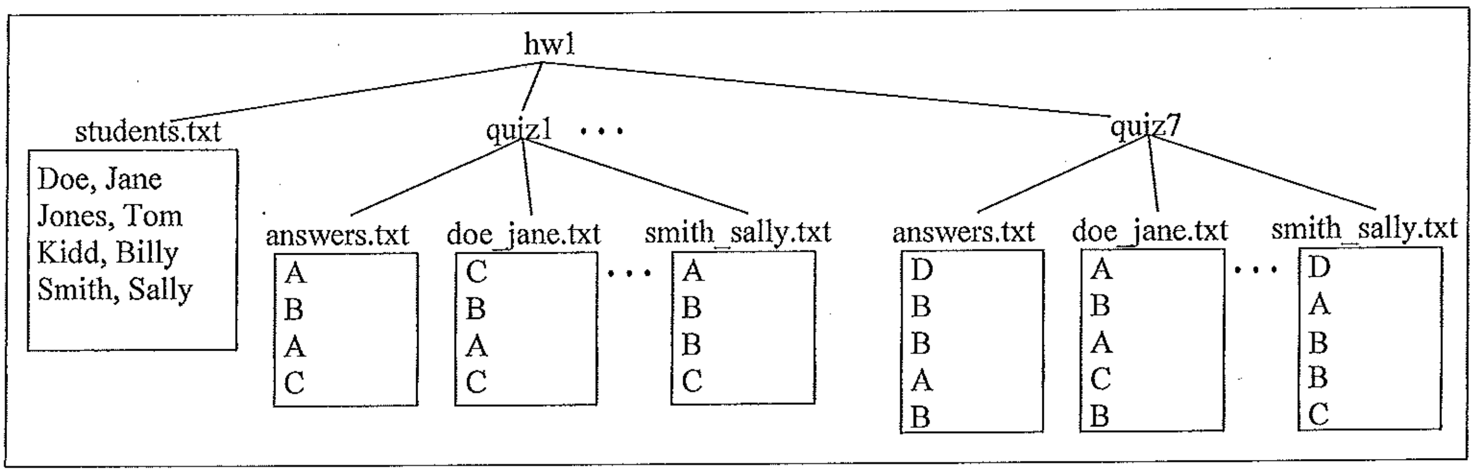
Homework #1 Data Structures Due: January 27 (Saturday at 11:59 PM)

Objects: Practice designing a program and "reviewing" Python file and os module usage. (NOTE: be sure to review the **example programs** `formattedOutput.py` **and** `changeDirectory.py` **found at:**
www.cs.uni.edu/~fienup/cs1520s18/homework/example_programs_hw1.zip)

Electronic Quiz Grader Program

The eLearning multiple-choice-quiz grader has broken down, so Professor Smart N. Lazy wants you to write a program (`quizGrader.py`) to grade the class's eLearning quizzes. After extracting the files from:
http://www.cs.uni.edu/~fienup/cs1520s18/homework/hw1.zip
you will find that the hw1 folder contains:

- `students.txt` - a text file containing the student names in the class
- one **or more** `quiz#` directories - each directory contains an `answers.txt` text file with the correct answers and text files for each student who took the quiz. The student file names are `lastname_firstname.txt`



Your program (called `quizGrader.py`) should run from inside the hw1 directory (i.e., develop it inside the hw1 directory) to generate a `gradeReport.txt` file that looks something like:

```
         Student Quiz Report

                  Total        Overall
Student         Quiz Points    Quiz %
------------------------------------------
Doe, Jane           30          71.4
Jones, Tom          40          95.2
Kidd, Billy         35          83.3
Smith, Sally        36          85.7

Points Possible     42
```

For extra credit, you can report more details (e.g., individual quiz scores for each student):

```
                             Student Quiz Report

                                                    Total      Overall
Student            Quiz 1  Quiz 2        Quiz 7  Quiz Points   Quiz %
--------------------------------- ... ------------------------------
Doe, Jane             3       5            2         30          71.4
Jones, Tom            4       6            5         40          95.2
Kidd, Billy           3       4            4         35          83.3
Smith, Sally          3       6            1         36          85.7

Points Possible       4       7            5         42
```

Text Files: Below is a summary of the important text-file operations in Python.

File Operations in Python		
General syntax	**Example**	**Description**
open(filename) open(filename, mode)	`f = open('data.txt', 'w')`	Modes: `'r'` read only; `'w'` write only; `'a'` append; `'r+'` both reading and writing. Default mode is `'r'`
f.close()	`f.close()`	Close the file to free up system resources.
loop over the file object	`for line in f:` `    print (line)`	Memory efficient, fast and simple code to loop over each line in the file.
f.readline()	`nextLine = f.readline()`	Returns the next line from the file. The newline ('\n') character is left at the end of the string.
f.write(string)	`f.write('cats and dogs\n')`	Writes the string to the file.
f.read( )	`all = f.read()`	Returns the whole file as a single string.
f.read(size)	`chunk = f.read(100)`	Returns a string of at most 100 (size) bytes. If the file has been completely read, an empty string is returned.
f.readlines()	`allLines = f.readlines()`	Returns a list containing all the lines of the file.
f.readlines(size)	`someLines = f.readlines(5000)`	Returns the next 5000 bytes of line. Only complete lines will be returned.

Below is a summary of the important file-system functions from the `os` and `os.path` modules in Python.

`os` Module File-system Functions	
General syntax	**Description**
`getcwd( )`	Returns the complete path of the current working directory
`chdir(path)`	Changes the current working directory to path
`listdir(path)`	Returns a list of the names in directory named path
`mkdir(path)`	Creates a new directory named path and places it in the current working directory
`rmdir(path)`	Removes the directory named path from the current working directory
`remove(path)`	Removes the file named path from the current working directory
`rename(old, new)`	Renames the file or directory named old to new

`os.path` Module File-system Functions	
General syntax	**Description**
`exists(path)`	Returns True if path exists and False otherwise
`isdir(path)`	Returns True if path is a directory and False otherwise
`isfile(path)`	Returns True if path is a file and False otherwise
`getsize(path)`	Returns the size in bytes of the object named path

NOTE: The initial "current working directory" is the directory where the program is located. Typically, it is useful to access files relative to the "current working directory" instead of specifying an absolute (complete) path. You can use the strings:

- `'.'` to specify the current working directory, e.g., `currentDirectoryList = os.listdir('.')`
- `'..'` to specify the parent of current working directory, e.g., `os.chdir('..')` which changes the current working directory to the parent directory

Data Structures (CS 1520) Lecture 2 Name:____________________

1. An alternative to functional-decomposition design is to use object-oriented design(OOD). For the following program, what objects would be useful and what methods (operations on the objects) should each support?

"Write a program to roll two 6-sided dice 1,000 times to determine the percentage of each outcome (i.e., sum boths dice). Report the outcome(s) with the highest percentage." (You only need consider the program's OOD)

Die : method : roll, getRoll, getSides, add(+) data attributes currentRoll #sides

TallySheet : method: incrTally, addColumn, data attribute : dictionary

2. Consider the Die and AdvancedDie classes from the Python Summary handout.

a) What data attributes of AdvancedDie are inherited from the parent Die class? _currentRoll

b) What new data attributes are added as part of the subclass AdvancedDie? _numSides

c) Which Die class methods are used directly for an AdvancedDie object? getRoll

d) Which Die class methods are redefined/overridden by the AdvancedDie object? __str__, __init__, roll

e) Which methods are new to the AdvancedDie class and not in the Die class? __eq__, __lt__, __gt__, getSides

f) If `die1` and `die2` are AdvancedDie objects, then the statement "`if die1 == die2:`" invokes the `__eq__` method of AdvancedDie with `die1` "passed" as `self` and `die2` passed as `rhs_Die`.

```
def __eq__(self, rhs_Die):
    """Overrides default '__eq__' operator to allow for deep comparison of dice"""
    return self._currentRoll == rhs_Die._currentRoll
```

What would the code be for AdvancedDie `__le__` method to allow for the "`if die1 <= die2:`" statement?

```
def __le__(self, rhs_Die):
    return self._currentRoll <= rhs_Die._currentRoll
```

g) Good software engineering practice is to include *precondition* and *postcondition* comments on each method/function where the:

- *precondition* - indicates what must be true for the method to work correctly. Typically, the precondition describes the valid values of the parameters. If the precondition is not satisfied, the method does not need to work correctly!
- *postcondition* - describes the expected state after the method has executed

Consider the AdvancedDie constructor:

```
class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides"""

    def __init__(self, sides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided."""

        Die.__init__(self)  # call Die parent class constructor
        self._numSides = sides
        self._currentRoll = randint(1, self._numSides)
```

```
if not isinstance(sides, int):
    raise TypeError("sides must be an integer")
if sides <= 0:
    raise ValueError("sides must be positive int")
```

What precondition and postcondition comments should we add?

Precondition: sides must be a positive integer

Postcondition: initialized current roll between 1 and sides

h) If a method/function has a precondition that is not met when invoked (e.g., `die1 = AdvancedDie("six")`), why should the method raise an error? 3.14

raise error where there error occurred

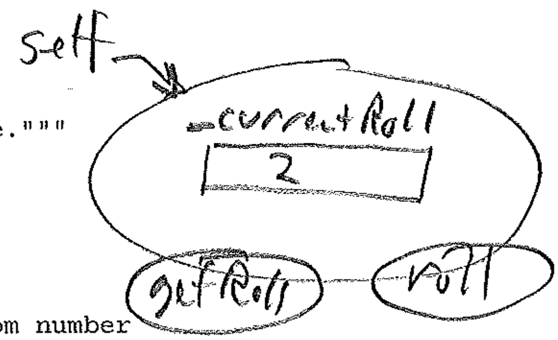
Classes: A *class* definition is like a blueprint (receipe) for each of the objects of that class.
A class specifies a set of data attributes and methods for the objects of that class

- The values of the data attributes of a given object make up its state
- The behavior of an object depends on its current state and on the methods that manipulate this state
- The set of a class's methods is called its *interface*

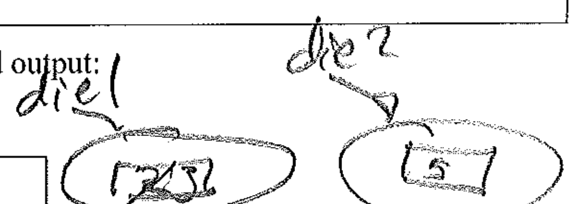
The general syntax of class definition is:

```
class MyClass [ ( superClass1 [, superClass2 ]* ) ]:
        '''Document comment which becomes the __.doc__ attribute for the class'''
        def __init__(self, [param [, param]*):
                '''Document comment for constructor method with self be referencing to the object itself'''
                #__init__body

        # defs of other class methods and assignments to class attributes

# end class MyClass
```

```
"""
File: simple_die.py
Description:  This module defines a six-sided Die class.
"""

from random import randint

class Die(object):
    """This class represents a six-sided die."""

    def __init__(self):
        """The initial face of the die."""
        self._currentRoll = randint(1, 6)

    def roll(self):
        """Resets the die's value to a random number
        between 1 and 6."""
        self._currentRoll = randint(1, 6)

    def getRoll(self):
        """Returns the face value of the die."""
        return self._currentRoll

    def __str__(self):
        """Returns the string representation of the die."""
        return str(self._currentRoll)
```



Consider the following script to test the Die class and its associated output:




```
# testDie.py - script to test Die class
from simple_die import Die

die1 = Die()
die2 = Die()
print('die1 =', die1)    #calls __str__
print('die2 =', die2)
print()
print('die1.getRoll() = ', die1.getRoll())
print('die2.getRoll() = ', die2.getRoll())
die1.roll()
print('die1.getRoll() = ', die1.getRoll())
print('str(die1): ' + str(die1))
print('die1 + die2:', die1.getRoll() + die2.getRoll())
```

```
>>>
die1 = 2
die2 = 5

die1.getRoll() =  2
die2.getRoll() =  5
die1.getRoll() =  3
str(die1): 3
die1 + die2: 8
>>>
```

Classes in Python have the following characteristics:

- all class attributes (data attributes and methods) are *public* by default, unless your identifier starts with a single underscores, e.g, `self._currentRoll`
- all data types are objects, so they can be used as inherited base classes
- **objects are passed by reference when used as parameters to functions**
- all classes have a set of standard methods provided, but may not work properly (__str__, __doc__, etc.)
- most built-in operators (+, -, *, <, >, ==, etc.) can be redefined for a class. This makes programming with objects a lot more intuitive. For example suppose we have two Die objects: `die1` & `die2`, and we want to add up their combined rolls. We could use *accessor methods* to do this:

  ```
  diceTotal = die1.getRoll() + die2.getRoll()
  ```

  Here, the `getRoll` method returns an integer (type int), so the '+' operator being used above is the one for ints. But, it might be nice to "overload" the + operator by defining an __add__ method as part of the Die class, so the programmer could add dice directly as in:

  ```
  diceTotal = die1 + die2
  ```

The three most important features of *Object-Oriented Programming* (OOP) to simplify programs and make them maintainable are:

1. *encapsulation* - restricts access to an object's data to access only by its methods
   ⇨ helps to prevent indiscriminant changes that might cause an invalid object state (e.g., 6-side die with a of roll 8)
2. *inheritance* - allows one class (the *subclass*) to pickup data attributes and methods of other class(es) (the *parents*)
   ⇨ helps code reuse since the subclass can extend its parent class(es) by adding addition data attributes and/or methods, or overriding (through polymorphism) a parent's methods
3. *polymorphism* - allows methods in several different classes to have the same names, but be tailored for each class
   ⇨ helps reduce the need to learn new names for standard operations (or invent strange names to make them unique)

Consider using inheritance to extend the Die class to a generalized AdvancedDie class that can have any number of sides. The interface for the AdvancedDie class are:

Detail Descriptions of the AdvancedDie Class Methods		
Method	**Example Usage**	**Description**
`__init__`	`myDie = AdvancedDie(8)`	Constructs a die with a specified number of sides and randomly rolls it (Default of 6 sides if no argument supplied)
`getRoll`	`myDie.getRoll()`	Returns the current roll of the die (inherited from Die class)
`getSides`	`myDie.getSides()`	Returns the number of sides on the die (did not exist in Die class)
`roll`	`myDie.roll()`	Rerolls the die randomly (By overriding the `roll` method of Die, an AdvancedDie can generate a value based on its # of sides)
`__eq__`	`if myDie == otherDie:`	Allows == operations to work correctly for AdvancedDie objects.
`__lt__`	`if myDie < otherDie:`	Allows < operations to work correctly for AdvancedDie objects.
`__gt__`	`if myDie > otherDie:`	Allows > operations to work correctly for AdvancedDie objects.
`__add__`	sum = myDie + otherDie	Allows the direct addition of AdvancedDie objects, and returns the integer sum of their current roll values.

`__str__`	Directly as:
	`myDie.__str__()`
	`str(myDie)`
	or indirectly as:
	`print myDie`
		Returns a string representation for the AdvancedDie. By overriding the __str__ method of the Die class, so the "print" statement will work correctly with an AdvancedDie.

Python Summary Name:____________________

Consider the following script and associated output:

```
# testAdvancedDie.py - script to test
AdvancedDie class
from advanced_die import AdvancedDie

die1 = AdvancedDie(100)
die2 = AdvancedDie(100)
die3 = AdvancedDie()

print( 'die1 =', die1 )     #calls __str__
print( 'die2 =', die2 )
print( 'die3 =', die3 )

print( 'die1.getRoll() = ', die1.getRoll())
print( 'die1.getSides() =', die1.getSides())
die1.roll()
print( 'die1.getRoll() = ', die1.getRoll())
print( 'die2.getRoll() = ', die2.getRoll())
print( 'die1 == die2:', die1==die2)
print( 'die1 < die2:', die1<die2)
print( 'die1 > die2:', die1>die2)
print( 'die1 != die2:', die1!=die2)
print( 'str(die1): ' + str(die1))
print( 'die1 + die2:', die1 + die2)

help(AdvancedDie)
```

```
die1 = Number of Sides=100 Roll=32
die2 = Number of Sides=100 Roll=76
die3 = Number of Sides=6 Roll=5
die1.getRoll() =  32
die1.getSides() = 100

die1.getRoll() =  70
die2.getRoll() =  76
die1 == die2: False
die1 < die2: True
die1 > die2: False
die1 != die2: True
str(die1): Number of Sides=100 Roll=70
die1 + die2: 146
Help on class AdvancedDie in module
advanced_die:

class AdvancedDie(simple_die.Die)
 |  Advanced die class that allows for
any number of sides
 |
 |  Method resolution order:
 |      AdvancedDie
 |      simple_die.Die
 |      __builtin__.object
 |
 |  Methods defined here:
```

Notice that the `testAdvancedDie.py` script needed to `import AdvancedDie`, but not the `Die` class.

The AdvancedDie class that inherits from the Die superclass.

```
"""
File:  advanced_die.py
Description: Provides a AdvancedDie class that allows for any number of sides
Inherits from the parent class Die in module die_simple
"""
from simple_die import Die
from random import randint

class AdvancedDie(Die):
    """Advanced die class that allows for any number of sides"""

    def __init__(self, sides = 6):
        """Constructor for any sided Die that takes an the number of sides
        as a parameter; if no parameter given then default is 6-sided."""

        Die.__init__(self)       # call Die parent class constructor
        self._numSides = sides
        self._currentRoll = randint(1, self._numSides)

    def roll(self):
        """Causes a die to roll itself -- overrides Die class roll"""
        self._currentRoll = randint(1, self._numSides)

    def __eq__(self, rhs_Die):
        """Overrides default '__eq__' operator to allow for deep comparison of Dice"""
        return self._currentRoll == rhs_Die._currentRoll

    def __lt__(self, rhs_Die):
        """Overrides default '__lt__' operator to allow for deep comparison of Dice"""
        return self._currentRoll < rhs_Die._currentRoll

    def __gt__(self, rhs_Die):
        """Overrides default '__gt__' operator to allow for deep comparison of Dice"""
        return self._currentRoll > rhs_Die._currentRoll

    def __str__(self):
        """Returns the string representation of the AdvancedDie."""
        return 'Number of Sides='+str(self._numSides)+' Roll='+str(self._currentRoll)

    def __add__(self, rhs_Die):
        """Returns the sum of two dice rolls"""
        return self._currentRoll + rhs_Die._currentRoll

    def getSides(self):
        """Returns the number of sides on the die."""
        return self._numSides
```

self → currentRoll 33, numSides 100

```
if not isinstance(sides, int):
    raise TypeError("sides must be an integer")

if sides <= 0:
    raise ValueError("sides must be a positive integer")
```

3. **General "Algorithmic-Complexity Analysis" terminology:**

<u>problem</u> - question we seek an answer for, e.g., "What is the sum of all the items in a list/array?"

<u>parameters</u> - variables with unspecified values

<u>problem instance</u> - assignment of values to parameters, i.e., the specific input to the problem

	0	1	2	3	4	5	6
myList:	5	10	2	15	20	1	11

sum: ?

(number of elements) n: 7

<u>algorithm</u> - step-by-step procedure for producing a solution

<u>basic operation</u> - fundamental operation in the algorithm (i.e., operation done the most) Generally, we want to derive a function for the number of times that the basic operation is performed related to the *problem size.*

<u>problem size</u> - input size. For algorithms involving lists/arrays, the problem size is the number of elements ("n").

<u>Big-oh notation (O( )</u> - As the size of a problem grows (i.e., more data), how will our program's run-time grow.

Consider the following `sumList` function.

```
def sumList(myList):
    """Returns the sum of all items in myList"""
    total = 0
    for item in myList:
        total = total + item
    return total
```

a) What is the basic operation of `sumList` (i.e., operation done the most) ? +

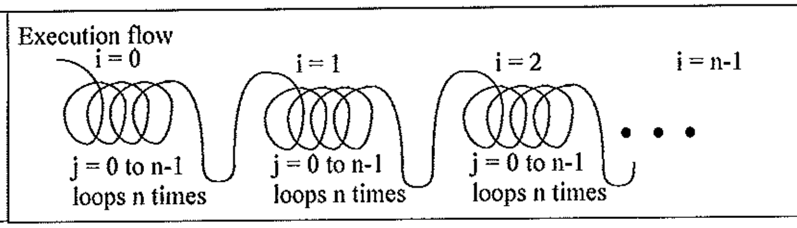
b) What is the problem size of `sumList`? length of myList, "n"

c) If n is 10000 and `sumList` takes 10 seconds, how long would you expect `sumList` to take for n of 20000? 20 sec

d) What is the big-oh notation for `sumList`? O(n) "linear"

4. Consider the following `someLoops` function.

```
def someLoops(n):
    total = 0
    for i in range(n):
        for j in range(n):
            total = total + i + j
    return total
```



a) What is the basic operation of `someLoops` (i.e., operation done the most)?

b) How many times will the basic operation execute as a function of `n`?

c) What is the big-oh notation for `someLoops`?

d) If we input n of 10000 and `someLoops` takes 10 seconds, how long would you expect `someLoops` to take for n of 20000?